

Step One: Document The Problem

baylor wetzel

14458 Alabama Ave S.
Savage, MN 55378
baylor@ihatebaylor.com

Abstract

This paper argues that the primary reason that AI problems persist is process-related, not technical and argues that significant progress can be made by documenting common problems, studying examples, using traceability grids, developing task-specific behavior capture tools and adhering to a common information capture format.

1. The Life of an AI Developer

Game development is hard work. And my belief is that, out of all the developers, the AI programmer has the toughest job. Not because the problem itself is hard but because the AI developer receives so little support. Until recently, there was no book or journal they could turn to for help. Their peers are discouraged from sharing information, worrying (quite legitimately) that AI is a competitive advantage. Unlike their teammates, they have no quantitative measures such as frames per second, no benchmarks similar to 3DMark, no standards equivalent to OpenGL or BSP and no standard tools such as Maya. AI developers don't even have a good description of the problem they're trying to solve. More often than not, each studio is forced to slay the AI problem alone, armed with little more than their personal experience. I believe there's a better way.

2. We Do Not Understand AI Problems

A recurring theme in this paper is that we do not know nearly enough about the problems we are trying to solve. What does good AI look like? What does an AI problem look like? What capabilities should an AI have? What are the most important capabilities? Which areas are being researched? Which areas are being ignored? How many AI problems are in an average game? How much progress has been made? How many problems are unique to a game? What percent of problems are common to a genre? Which problems are the most important? The answer to all of these questions is, we simply don't know.

In life, we learn by experiencing varying situations first and constructing models second. We divide experiences into categories, look for patterns and construct beliefs, but only after we are familiar with the details. When a solution is created independent of the problem, the solution is not

grounded in reality. Yet a review of several dozen peer-reviewed papers turned up less than a handful that gave a single concrete example. Only three attempted to define general areas of game AI. And only one attempted to analyze real problems. And it was written by a military officer bemoaning the lack of realistic work being done by modeling and simulation students (Marshall et al 2003).

Does anyone know what an AI problem looks like? An Internet search turned up one group that does – customers. Their laughs, frustration and disbelief can be found scattered across newsgroups, forums, game reviews, FAQs and strategy guides (see, for example, (Civilization Fanatic Center 2004)). The bad news for researchers is that these comments are hidden across the Internet, making the use and just-in-time collection of this information difficult.

In an effort to further my own research, I created a database to store examples of common game AI problems. Over the last year, I have collected several dozen problems which I have divided into 14 basic categories (learning, decision making, etc.) and further subdivided into several subcategories (discrimination learning, relationship learning, learning by observation, etc.). Studying these problems lead to many of the beliefs that follow.

3. AI Problems Are Design Flaws, Not Bugs

User forums often debate topics such as “which game has the worst AI?” (e.g., Slashdot 1999). The problems most ridiculed are almost never specific to one game.

For example, in many first person shooters, the monsters are in a frozen state until they see the player. If you can see a deactivated monster and it does not see you, you can attack the monster and it will scream, thrash around and eventually die, but it will not attack, hide or run away. Players who noticed this problem in Doom were quick to look for (and find) this exploit in later games such as Heretic, Quake and Half-Life.

A similar problem exists in the Baldur's Gate series of games. Monsters are in a frozen state until they are drawn on the screen. They cannot act, but they can take damage. Many FAQs and strategy guides recommend using the area-effect tactic. The player gets close to the unseen

enemy and then, because they cannot target an off-screen character, uses a spell that affects an area rather than a monster (e.g., cloudkill, ice storm, etc.). This is repeated until the enemies die. If the player's team runs out of spells, they go to sleep, memorize new spells and tries again the next day. It's not like the monster's going anywhere.

Problems like these are well known, exist in multiple games and affect some very respectable titles. For example, the "enemies don't react to pain" problem exists in both Half-Life and Baldur's Gate II: Shadows of Amn, both of which won Game of the Year and both of which were praised for their cutting-edge AI.

There are many problems like these and they span games, developers and genres. For example, groups of agents cannot move through doors in Counter-Strike (a first person shooter) and Planescape: Torment (a computer role-playing game). Agents continue to attack enemies with weapons they know will not work in Baldur's Gate, Civilization and Total Annihilation. The computer will consistently use a path they it knows is blocked by undefeatable enemies in Ages of Empire, Unreal Tournament, Civilization, Counter-Strike, Warcraft, Hitman and Myth. Running in front of an ally's line of fire, inability to see through windows, choosing wildly inappropriate spells, picking fights that it knows cannot be won, moving and attacking as individuals rather than as a group and others can be found in most games.

4. AI Problems Are Persistent

If there is one thing more frustrating than seeing the same problem over and over, it's seeing it year after year. It's almost certainly true that the average developer does not have enough time during development to find and fix all AI issues, but when a game's known problems are older than its customers, there's an issue.

Consider the pond fish. A pond fish is a navy built in a small body of water, often only one tile in size. Players of Civilization, 1991's Game of the Year, joked about it, gave it a nickname and created Web sites to share screen shots of it (e.g., (Civilization Fanatic Center 2002)). Despite the problem's high profile, the pond fish was still there in 2001's Civilization III and 2002's award winning Heroes of Might & Magic IV.

These problems have stood the test of time. The time span between Baldur's Gate and Icewind Dale 2, games which allow you to attack enemies without fear of reprisal, is three years. For Doom and Half-Life, which also allow free attacks, it's five. The span between the first pond fish and the latest is 11 years, making it four years older than my oldest child, this game's number one fan. And these are conservative numbers. They are based only on incidents I

have personally witnessed. If customer reports are to be believed, these problems are actually much older.

5. AI Problems Are Solvable

Earlier we said that in many first person shooters, enemies sometimes don't mind being shot. Because an enemy screams and plays a damage animation when it is shot, we can assume that code already exists to detect pain. The agent's actions are normally determined by the value of a single state variable. My assumption is that this problem can be solved simply by adding a line to the `damageTaken` event that sets the enemy's state to `Attacking`. Alternately, you could make the agent invulnerable while `Disabled`. Either solution is simple. So why does this problem still exist?

A common argument is that the AI doesn't get enough processing time. The solution, they argue, is to build an AI coprocessor. Yet the change discussed here takes only a few clock cycles and it's unclear how an AI coprocessor would make this better.

Another common argument is that game developers lack a formal AI education. Yet it is hard to imagine that the above fix requires a PhD.

It is often argued that AI developers simply don't have enough time. True enough. So a developer can be forgiven for missing the problem the first time around. But after five years and a dozen games?

A fourth explanation is that an AI developer cannot predict every single event that will happen in a game. That might be true. But how hard is it to predict that a player might attack his enemy? How hard is it to discover a problem that players have been reporting for years?

This problem has a simple solution. Perhaps it's unique. Without a database of problems, it's hard to know. My personal incident database suggests that it is not. Let's examine a few more.

The area effect tactic described earlier is solved the same way as the problem above. How about when enemies don't notice that the person next to them has died? Since they already respond to sound, have the victim scream. Agents keep taking the same deadly path? Overlay a grid, track the death count in each cell and use it as a cost for A*. Pond fish? Since these are tile games, you can quickly measure the size of a lake before building a navy. Agents keep using the same ineffective weapon? In the code that plays "My weapon did no damage!", have them switch weapons.

And so on and so on. There are issues that require some thought (e.g., avoiding bottlenecks when groups move through a door), but the majority appear to be fixable, even when development time and processing power are short.

Problem Area: action selection
 AI Type: decision making,
 learning (secondary),
 personality (secondary)
 Detail Level: mid-level
 Technique: Matching Law
 Assumptions: Options are relatively equal
 Example Uses: sports: choosing a shot type
 FPS: choosing a weapon
 RPG: choosing a spell
 RTS: choosing a build unit type

Game Example

A wizard is 30 meters away from a group of orcs. He has three third level spells he can use - flameshield, iceblade and stonestorm. Question: which spell should the wizard cast?

Assume that the wizard has successfully hit his enemies 3/10 times with flameshield, 2/4 times with iceblade and 6/7 times with stonestorm.

$r(\text{flameshield}) = 3/10 = 0.3$ (30%)
 $r(\text{iceblade}) = 2/4 = 0.5$ (50%)
 $r(\text{stonestorm}) = 6/7 = 0.86$ (86%)

Explanation

[truncated for space]

Variables

$RA/RA+RB = b(rA/rA+rB)^s$ = matching law

RA = Rate of response for option A.
 How often option A is chosen.
 This is a counter

rA = Rate of reinforcement for option A.
 The percentage of time choosing option A has lead to a good result

$rA/rA+rB$ = Relative rate of response for option A.

b = Response bias.

relative $r(f) = .3/.3+.5+.86 = 0.18$ (18%)

relative $r(i) = .5/.3+.5+.86 = 0.30$ (30%)

relative $r(s) = .86/.3+.5+.86 = 0.52$ (52%)

So the wizard would cast stonestorm 52% of the time, iceblade 30% and flameshield 18%

Limitations

- As stated, does not support learning by observation. It can be added though

Notes

- Bias and sensitivity are hard coded

Figure 1. A sample solution case using a proposed solution template (edited for brevity)

6. A Good Start: Build a Foundation

So we know that many AI problems that are easily solved are widespread, exist in top games and are old enough to go to school. What should we do? In my opinion, we start by collecting, documenting and sharing information.

A. Build a Database of Examples

I believe that the best way to understand a problem is to study examples of it. The examples should be from high-quality commercial games and should be replicatable. To distinguish between bugs and AI design flaws, the examples should be representative of problems that exist in multiple games. Examples should be detailed. Many game AI problems have constraints and catches that will not be understood if the example is not sufficiently detailed.

In my database, I include the game's name and publication date, a detailed description of the problem and the steps necessary to recreate it. Notes and images are optional.

B. Analyze Problems

i. Classify Incidents

After you have a realistic set of detailed examples, you can look for problems. There are a few steps here. First, read through each example and decide what problem it illustrates. The problem should be fairly low level, such as inability to follow the player up stairs. Second, group examples together by problem. Group those problems into

more general groups, building a problem hierarchy. Repeat as needed. As an example, suppose you have several instances where an enemy stops chasing a player due to a ladder, door or elevator. These three problems roll up to the group "inability to navigate dynamic terrain" which rolls up to "navigation".

ii. Look for Patterns

There are a few reasons to create a taxonomy of problems. First, it is easier to find patterns when the data can be analyzed at different levels of abstraction. Second, grouping makes it easier to manage large amounts of data.

A third reason is that some problems are common across categories. For example, selecting an opponent, weapon, spell, strategy, unit, utterance and route all require the ability to make a choice. Recognizing this, a developer might create a single solution that allows an agent to choose the best avenue of approach, spot to hide and weapon to use. The benefit is quicker development and potentially easier maintenance.

A fourth reason is that studying the allocation of real world incidents can help a researcher understand the size and importance of each problem area. This helps researchers decide which areas are most in need of study.

iii. Reuse the Information

Incidents can be used as test cases, benchmarks and scenarios for behavior capture tools. High level problem categories can be used as a checklist during design and testing to make sure that no relevant AI area is forgotten.

Category	Problem	Scenario	Solution	Test Case	Verification
1. Learning	A. choose best item	1. Choose sword task	Matching Law	Choose sword test	choice capture tool
1. Learning	B. learn relationship	1. Navigate sniper zone task	Rescorla-Wagner	Navigate sniper zone test	navigation capture tool
1. Learning	C. generalization	1. Rate new fire spell task	discrimination training	Rate new fire spell test	choice capture tool

Table 1. Sample Traceability Grid

C. Document Solutions

i. Collect Algorithms, Data Structures and Processes

Once a problem has been solved, the solution should be captured. Solutions are stored independent of the problems because a single solution might apply to many problems. It is my personal opinion that a solution should not be added to the database until it has been used in the field.

While a solution might be an algorithm, much (if not most) of human intelligence is due to good data representations. The solution database should therefore contain data describing useful data structures. And because a solution is sometimes a design process rather than a specific artifact, the database should store those too.

As an aside, there are only a few good papers and books relevant to game AI, but the topic has gotten more popular recently and as a result more work is being published. The military simulation community has produced several good papers. Search for “computer generated forces” and “behavioral representation in modeling and simulation” to find them. Finally, if you want agents to act like humans, consider learning about psychology (especially behavioral) and ethology (animal behavior).

ii. Determine Strengths and Weaknesses

In home construction, certain tasks are best solved with certain tools. Hammers, saws, wrenches and caulk, the best approach is a toolkit, not a single, all-powerful tool. The mind works in the same way. So should your AI.

As an example, the Matching Law models how humans choose between two options of roughly equal value but does not explain how they choose between unequal values. BSPs are good for storing navigation data but are a poor choice for storing a spell list. A Bounding Overwatch maneuver (SOG Co-op Squad 2004) makes sense for a SWAT team but not for zombies. When documenting a solution, make sure to list the problems the technique is and is not good for, items to use in addition to the technique, alternatives and any important information about implementation (memory consumption, speed, etc.).

iii. Use A Standard, Useful Format

The value of information is significantly determined by the format of the data. For an example of the format I use, see Figure 1. The solution case contains 11 fields – Problem Area, AI Type, Detail Level, Technique, Assumptions, Example Uses, Explanation, Variables, Game Example, Limitations and Notes.

Problem Area describes what type of problem this solution addresses. AI Type describes the functional areas that can use this solution (in the example, decision making, learning and personality). Some solutions will deal with very low-level problems while others will address higher level issues. The detail level of the problem being solved is stored in Detail Level. Technique is the name of the solution, Assumptions lists assumptions made about the problem and Example Uses gives examples of different problems in different genres where the solution can be used.

The Explanation section explains the technique and any information useful to the person using it. The Variables section defines each variable used in the technique, explains why it is needed, gives hints for implementation and explains how the user obtains the input values.

Finally, the Game Example section walks through a realistic example and shows how the formula is used and how the inputs are obtained. When a formula can be used for multiple purposes, an example is given of each.

My average solution case is roughly two pages long, detailed enough to be useful, short enough to be readable.

D. Tie It All Together

The value of data depends on how it’s used. To see the relationship between a problem and a solution, I view the data as a six-column traceability grid (for an example, see Table 1). Done well, it should make clear why a certain technique exists, whether a problem is understood (i.e., has examples), whether there are known solutions to a problem, how we know the solution is correct and, in general, why we’re doing what we’re doing.

The six columns I track are Category, Problem, Scenario, Solution, Test Case and Verification. The columns are arranged in the order that I work on them. Only the Category field is required, although I get nervous about a category if I cannot quickly think of a specific problem and example. The meaning of each column is described below.

The top group in the problem hierarchy is displayed in Category, in this example, learning. The leaf node of the problem tree is shown in Problem. In the example, these are learning a preference for an option, learning the relationship between a set of items and generalization.

Specific examples of a problem are listed in Scenario. A Scenario is similar to a scientific experiment. The terms must be objective enough that all readers agree on their meaning and there must be enough detail that the scenario can be replicated. The Game Example section of Figure 1 provides a good example.

Solution refers to a solution case. The format of this case was discussed earlier.

A Test Case is a Scenario combined with the expected answer. AI test cases are slightly more complicated than traditional unit tests because an agent's actions normally depend heavily on that agent's "learning history" (i.e., experience) and because the output is often one of a weighted set of options (e.g., given a specific learning history, an agent will advance 50% of the time, pass 35% and shoot 15%). These are not particularly difficult issues but space constraints require that a detailed discussion of testing wait for a future article.

The test case gives the answer to the question posed by the scenario. A fair question to ask is, where does the test case get its answer from? Personally, I take my answers from psychology books, but how do I know the information I've gleaned from them is correct? The Verification field describes how we know a given solution is correct.

There are a number of ways to determine what answer a player would give. The most obvious is to allow the user to play the game and log their responses. My preference is to write task-specific behavior capture tools because they allow me to isolate the important variables. Again, room does not permit an in-depth discussion here.

E. Share the Results

My final suggestion is that this information be a community asset. My preference is for a Web-based system that allows anyone to add notes, similar to PHP.net's PHP API (Bakken & Schmid 2004). It is realized that studios might not choose to share their latest techniques, but there should be little reason not to contribute to the other portions.

As mentioned earlier, I have already started work on such a database. However, because I am not currently affiliated with any institution, I have chosen not to publish the temporary address of my work in this paper. Should there be sufficient interest, I will work on finding a permanent home for the data.

7. Concluding Thoughts

The process argued for here is not a magic bullet and does not promise to make problems magically disappear. The process of collecting, documenting and analyzing information is labor intensive, time consuming and requires cooperation from experts in a variety of fields. Gains will be incremental and the problems solved first will be those problems which the participants know the answers to or which they choose to tackle. Despite this, a better understanding of game intelligence will hopefully bring us closer to the critical mass required to make rapid, continuous progress. Other fields already take for granted that basic information is collected, analyzed and shared. It is time for the field of game intelligence to join them.

8. Acknowledgements

The idea of collecting common problems for the benefit of the wider community was inspired by two papers, "Fabulous Graduate Research Topics in CGF" (Marshall et al 2001) and "Toward A Human Behavior Models Anthology for Synthetic Agent Development" (Silverman et al 2001). I also wish to thank Randy Jones, John Laird, Frank Ritter and Tim Wilkin, none of whom pulled their punches. This paper, like so many others, is better for it.

References

- Bakken, S. and Schmid, E. April 29, 2004. PHP: A simple tutorial – Manual. <http://www.php.net/manual/en/tutorial.php>.
- Civilization Fanatic Center. Sept 2, 2002. Civilization III: Screenshot of the Day. <http://www.civfanatics.com/sotd/sotd1-30.shtml>.
- Civilization Fanatic Center. Jan 15, 2004. List of AI Stupidities in Civilization II. <http://www.civfanatics.com/civ2ai.shtml>.
- Marshall, H; Harris, T; Law, D and Rieger, L, 2001. Fabulous Graduate Research Topics in CGF, Improving the Army Graduate Program. *Proceedings of the 10th Conference On Computer Generated Forces and Behavioral Representation*.
- Silverman, B; Might, R; Dubois, R; Shin, H; Johns, M and Weaver, R. Mar 2001. Toward A Human Behavior Models Anthology for Synthetic Agent Development. *Proceedings of the 10th Conference On Computer Generated Forces and Behavioral Representation*.
- Slashdot. Jun 24, 1999. State of Computer Game AI. <http://slashdot.org/articles/99/06/24/1545204.shtml>.
- Special Operations Group Co-op Squad. May 24, 2004. Tactics. *SOG Co-op Squad – Ghost Recon Headquarters*. <http://trox.junglescene.com/GhRWebsite/Tactics.html>.